John von Neumann Institute for Computing

Parallel I/O Aspects in PIMA(GE)^2 Lib

Andrea Clematis, Daniele D'Agostino, Antonella Galizia

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 441-448, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Parallel I/O Aspects in $\mathrm{PIMA(GE)^2}$ Lib

Andrea Clematis, Daniele D'Agostino, and Antonella Galizia

Institute for Applied Mathematics and Information Technologies
National Research Council, Genoa, Italy
E-mail: {clematis, dago, antonella}@ge.imati.cnr.it

Input/Output operations represent a critical point in parallel computations, and often results in a bottleneck with a consequent reduction of application performance. This paper describes the parallel I/O strategy applied in $\mathrm{PIMA(GE)^2}$ Lib, the Parallel IMAGE processing GEnoa Library. The adoption of a parallel I/O results in an improvement of the performance of image processing applications developed using the library. To achieve this goal we performed an experimental study, comparing an MPI 1 implementation of a classical master-slave approach, and an MPI 2 implementation of parallel I/O. In both cases we considered the use of two different file systems, namely NFS and PVFS. We show that MPI 2 parallel I/O, combined with PVFS 2, outperforms the other possibilities, providing a reduction of the I/O cost for parallel image processing applications, if a suitable programming paradigm for I/O organization is adopted.

1 Introduction

The scientific evolution allows the analysis of different phenomena with great accuracy and this results in a growing production of data to process. Parallel computing is a feasible solution, but a critical point becomes an efficient I/O management. This issue may derive from an hardware level and/or from a poor application-level I/O support; therefore I/O performance should be improved using both parallel file systems and effective application programming interface (API) for I/O appropriately[1].

In this paper we focus on these aspects for parallel image processing applications. We developed $\mathrm{PIMA(GE)^2}$ Lib, the Parallel IMAGE processing GEnoa Library; it provides a robust implementation of the most common low level image processing operations. During the design of the library, we look for a proper organization of I/O operations, because of their impact on application efficiency. In fact, a parallel application interacts with the underlying I/O hardware infrastructure through a *software stack*, depicted in Fig. 1; the key point is to enable a proper interaction between the different levels[2].

In particular a parallel image processing application developed with $\mathrm{PIMA(GE)^2}$ Lib exploits a software level, or *I/O Middleware*, aimed to perform I/O operations using MPI. The I/O Middleware imposes the logical organization of the parallel processes and the I/O pattern to access data. It also interacts with the file system, e.g. PVSF, NFS, that in turn effectively exploits the I/O hardware, managing the data layout on disk. Thus a proper use of the I/O API provided by MPI leads to a more efficient management of the I/O primitives of the file systems.

The adoption of a parallel I/O in scientific applications is becoming a common practice. Many papers provide a clear state of the art of the problem; an in-depth analysis of I/O subsystems, and general purpose techniques to achieve high performance are proposed[1,5]. They mainly suggest the use of specific software to enable parallel access to the data. It is actually obtained considering parallel file systems combined with scientific data library, in

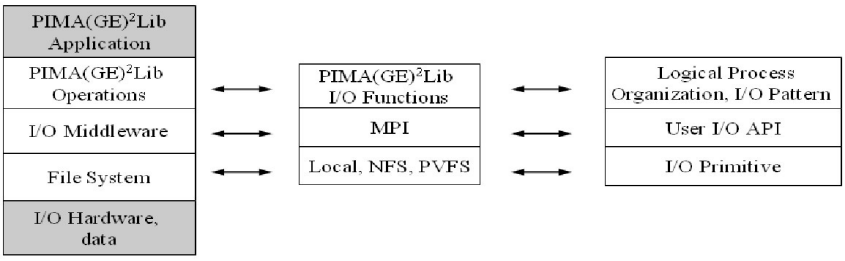

Figure 1. *Software stack for parallel image processing applications. Starting from the left, it is explained how an application interacts with data and I/O infrastructure, the tools used in* $\mathrm{PIMA(GE)}^2$ *Lib, and the roles of each software level.*

order to exploit their optimization policies. The user has to consider the strategy that better fits with the application requirements and the available I/O subsystem.

The proposed solutions have been adopted in different works, for example in visualization problems managing great amount of data[6], in simulations using particle-mesh methods[7], in biological sequence searching[8], in cosmology applications based on the adaptive mesh refinement[9]. Surprisingly, such strategies have not been sufficiently considered in the image processing community, although an increasing attention is paid to parallel computations. In fact it is possible to find different and actual examples of parallel libraries, ParHorus[10], PIPT[11], EASY-PIPE[13] and Oliveira et al.[12]. However they do not consider a parallel I/O, and apply a master-slave approach during the data distribution.

Starting from these remarks, we performed a case study about different approaches in the imaging community, where these aspects received only a little attention. We made several tests of the logical organization in I/O operations to determine the most efficient strategy to apply in $\mathrm{PIMA(GE)}^2$ Lib. To achieve this goal we compared a master-slave approach implemented with MPI 1, and a parallel I/O using the functionalities of the MPI-IO provided by MPI 2. In both cases we tested the interaction with the most common file systems for parallel and distributed applications, that are PVFS, Parallel Virtual File System[3], and NFS, Network File System[4], both open source. We show that MPI 2 parallel I/O, combined with PVFS 2, outperforms the other possibilities, providing a reduction of the I/O cost for parallel image processing applications. More in general a parallel I/O approach is effective in the image processing domain.

The paper is organized as follows: in the next Section a brief presentation of the $\mathrm{PIMA(GE)}^2$ Lib is given, including an overview of both I/O approaches we tested. In Section 3 we analyse the experimental results. The conclusions are outlined in Section 4.

2 A Brief Overview of the $\mathrm{PIMA(GE)}^2$ Lib with Different Parallel Organizations

The Parallel IMAGE processing GEnoa Library, shortly $\mathrm{PIMA(GE)}^2$ Lib, is designed with the purpose of providing robust and high performance implementations of the most common low level image processing operations, according to the classification provided

in Image Algebra[14]. The library has been implemented using C and MPICH, and allows the development of compute intensive applications, achieving good speed up values. The operations are performed both in a sequential and in data parallel fashion, according to the user requirements, on 2D and 3D data sets. The parallelism in $\mathrm{PIMA(GE)}^2$ Lib is hidden from the users through the definition of an effective and flexible interface, that appears completely sequential[15]. Its aim is to shield the users from the intrinsic complexities of a parallel application. An optimization policy is applied in the library in order to achieve good performance; the optimization aspects are transparent as well.

Let us focus on the I/O aspect of the library, with a description of the possible strategies to perform the I/O operations. Typically image processing applications acquire and produce data stored on files.

In order to avoid many small noncontinuous accesses to a possibly remote disk made from multiple processors, the classic logical organization for data distribution in parallel applications is the master-slave one. It means that a process, the master, entirely acquires

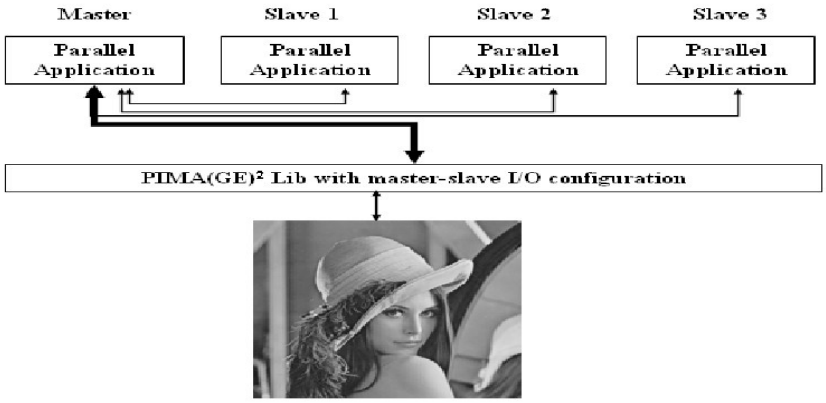

Figure 2. *Master-slave approach in accessing data. Only the master accesses the file, and sends portions of the image to the other processes.*

data and distributes them among the other MPI processes, the slaves, according to the I/O pattern. A specular phase of data collection is necessary for the output operations. Therefore the master is in charge of collecting/distributing data and performing I/O to a single file through a sequential API. This behaviour is depicted in Fig. 2.

However the data collection on a single process results in a serialization of the I/O requests and consequently in a bottleneck, because of the time the master spends in loading the entire data set and in sending the partial data to each process. A specular situation occurs for the Output operations. The waiting time for the distribution or the collection of data increases with the data set size. In case of huge data sets, the I/O execution may result very inefficient or even impossible. A further problem is the possibility to overwhelm the memory capacity, with the consequent necessity of exploiting the virtual memory.

An alternative approach to perform I/O operations and avoid unnecessary data movements is provided by a parallel access to the data file. It means that all processes perform

I/O operations, but each of them acquires or produces its specific portion of data. The situation is represented in Fig. 3.

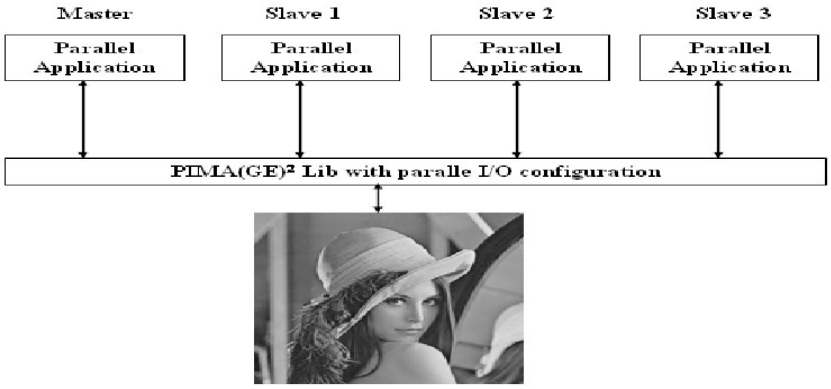

Figure 3. *The parallel I/O approach in accessing data. All the processed access data, and acquire their portion.*

However the partial data of each process may correspond to non contiguous small chunks of global data; it implies that each process accesses the I/O file to load small pieces of information non contiguously located. This situation worsens the performance even if sophisticated parallel file systems are used. In fact it is not possible to exploit file system optimization policy, since it is mainly designed to efficiently support the parallel access to large chunks of a file or of different files.

Another important point is given by the mismatch between the data access patterns and the actual data storage order on disk. This topic could be stressed if data are distributed on different machines. In this case, chunks of data requested by one processor may be spanned on multiple I/O nodes; or multiple processors may try to access the data on a single I/O node, suffering from a long I/O request queue. However combining the use of proper file systems, and API for I/O in a suitable way, avoids such situations and enables to effectively exploit a parallel I/O. In fact in this way it is possible to span data cleverly and remove the long queue in data acquisition.

3 I/O Experimental Results

We performed an experimental study about I/O organization in image processing, comparing the approaches already described, and fixing as I/O pattern a block partition. The master-slave approach was implemented through MPI 1, the parallel I/O using the functionalities of the MPI-IO provided by MPI 2. We are interested in the analysis of the I/O scalability and the impact of file systems on it; therefore, we measured how the growth of the number of processes affects the execution time in each case, and how each I/O approach interacts with different file systems, considering the use of PVFS 2, and NFS. In this paper we are not interested in the evaluation of the $\mathrm{PIMA(GE)}^2$ Lib performance, thus we do not consider other operations of the library.

3.1 Experimental Conditions

Tests have been performed on a Linux Beowulf Cluster of 16 PCs, each one equipped with 2.66 GHz Pentium IV processor, 1 GB of Ram and two EIDE 80 GB disks interfaced in RAID 0; the nodes are linked using a dedicated switched Gigabit network.

The experimental results were collected using the Computed Tomography (CT) scan of a Christmas Tree (XT)[a], and the CT scan of a Female Cadaver (FC)[b]. We considered such data sets because of their sizes; the XT data set can be considered a medium size data set, the partial image size varies from 499.5 MB to 31.2 MB considering respectively 1 and 16 processes, while the FC data set is a quite large size data set, and the partial image size varies from 867 MB to 54.2 MB in the same conditions.

With respect to the utilized software tools, let us provide few concepts about MPI-IO, and two widely used file systems for file sharing using clusters NFS and PVFS2.

NFS was designed to provide a transparent access to non local disk partitions. It means that, if we consider the machines of a local area network that mount the NFS partition on a specific directory, they are able to share and access all the file of that directory. Therefore a file sitting on a specific machine, looks to the users on all the machines of the network, as if the file resides locally on each machine.

PVFS stripes file data across multiple disks in different nodes in a cluster. It allows multiple processes to access the single part of the global file that have been spanned on different disks concurrently. We considered the second version of PVFS, (PVFS2), that allows the exploitation of fast interconnection solutions and the minimization of bottleneck due to the retrieving of metadata regarding the file to acquire or produce.

MPI-IO permits to achieve high performance for I/O operations in an easy way. Indeed it enables the definition of the most common I/O patterns as MPI derived data types, and in this way permits parallel data accesses. We considered the use of ROMIO[16,2], a high-performance, portable implementation of MPI-IO distributed with MPICH. ROMIO contains further optimization features such as Collective I/O and Data Sieving. These aspects have been implemented for several file systems, including PVFS2, NFS.

3.2 The Master-Slave Approach

We implemented the data partition through a sequential read operation performed by the master that immediately after scatters partial data to the slaves. The execution time (in seconds) are presented in Figure 4(a) for XT and in Figure 4(b) for FC data set.

In the tests involving NFS, we consider two different nodes of the cluster to store data and to run the master. In such situation, the data are not located on the same machine of the master, therefore we actually exploit the use of NFS in the data access. However we tested also a slightly different situation, i.e. the master has the data locally. In fact managing data through a remote file system, we have to take into account the overheads due to the latencies deriving from the file system and from the transmission time. We verified that considering XT data set the I/O using the local disk requires 0.9 seconds, and through NFS 1.2; while considering FC we have 1.4 and 4.6 respectively.

[a] The XT data set was generated from a real world Christmas Tree by the Department of Radiology, University of Vienna and the Institute of Computer Graphics and Algorithms, Vienna University of Technology,

[b] The FC data set is a courtesy of the Visible Human Project of the National Library of Medicine (NLM), Maryland.

In the master-slave approach, the striping of the data among multiple disks obtained with PVFS represents a drawback. Actually, since there is only one sequential access to the data file, the master process has to acquire data by accessing small parts on multiple disks. This represents an useless time consuming step. Indeed in Figure 4 we can see that on both data sets, the I/O performance do not scale and the execution time is almost constant. It is due to the overhead related with the use of the file system to access remote part of the data. It results higher than the time spent to send/receive data, in fact we do not verify a significant variation in the execution time even with the growth of the number of processes, i.e. the number of send/receive operations to perform.

On contrary, the use of a single disk through NFS represents on average the best solution, despite the data set size may really affect the performance. As it is possible to see in Figure 4, the use of NFS performs better than PVFS when we consider a medium size data set; but when we manage a large data set the use of PVSF leads to better performance when the number of process increases.

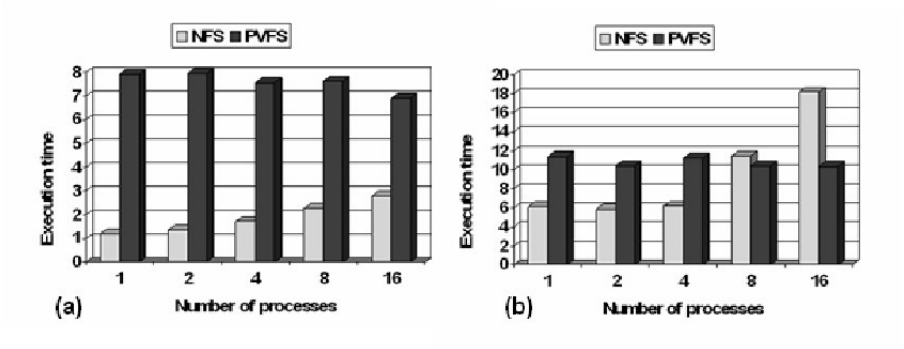

Figure 4. The execution time (in seconds) of the I/O operations using the master-slave approach on the XT data set (499.5 MB) (a) and the FC data set (867 MB) (b)

In fact in this case, even if data file is accessed through NFS, the data set is placed on a single disk of the cluster; that implies a lower overhead due to the file system. On the other hand, when the number of processes grows, the execution times suffer from the communication overheads. Considering up to 4 processes, the time could be considered similar to the sequential case; however the performance are really deteriorated if an higher number of processes is considered.

3.3 Parallel I/O

We implemented the parallel I/O using the collective I/O features and the derived datatypes provides by MPI 2. The execution time (in seconds) are presented in Figure 5(a) considering the XT data set, and Figure 5(b) for the FC data set. We can see that the parallel I/O combined with the use of PVFS outperforms the use of NFS and both master-slave solutions.

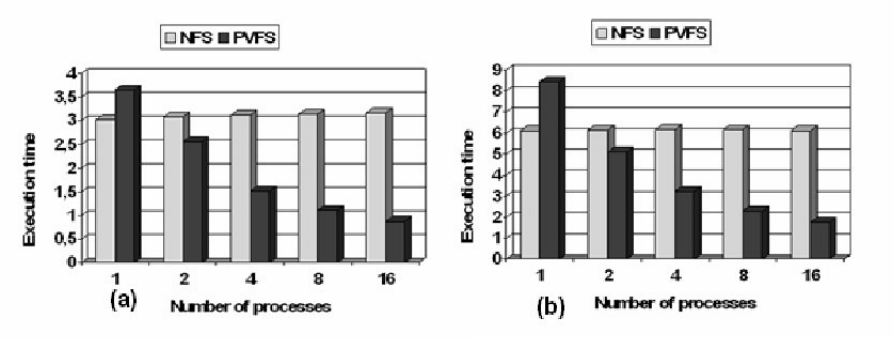

Figure 5. The execution time (in seconds) of the I/O operations using the MPI2 functionalities on the XT data set (499.5 MB) (a) and the FC data set (867 MB) (b)

It is mainly due to a combination of the two-phase I/O strategy adopted by the collective I/O operations, and the striping of the data among multiple disks performed by PVFS. In fact through the collective I/O operations, we significantly reduce the number of I/O requests that would otherwise result in many small non-contiguous I/O operations. By using the MPI derived data-types, the file data is seen as a 3D array, and the part requested by each process as a sub-array. When all processes perform a collective I/O operation, a large chunk of contiguous data file is accessed. The ROMIO implementation on PVFS2 is optimized to efficiently fit the I/O pattern of the application with the disk file striping. Thus the possible mismatches between the I/O pattern of the application and the physical storage patterns in file are minimized.

In Figure 5, we can see that MPI 2 and PVFS scales well with the number of processes, since in this case we effectively exploit data access to multiple disks. Instead it does not happen using NFS, since NFS was not designed for parallel applications that require concurrent file access to large chunk of files. Therefore as the number of processors and the file size increase, the use of NFS leads to an important bottleneck due to the serialization of the I/O operations. Actually the execution time is almost constant, although the number of processes increases.

4 Conclusions

The efficient acquisition and production of data is a major issue for parallel applications; this is of particular importance in the imaging community where these aspects received only a little attention.

In this paper we present how we tackled the problem in the design of $\mathrm{PIMA(GE)}^2$ Lib. Our solution is based on the use of the more sophisticated I/O routines provided by MPI2. This work represents an experimental study about the adoption of a parallel I/O, obtained comparing its performance with that achieved by the classical master-slave approach. The results demonstrated the effectiveness of parallel I/O strategy. In this manner we improve the overall performance of an application developed using the library. Furthermore, at the best of our knowledge, $\mathrm{PIMA(GE)}^2$ Lib is one of the few examples of image processing

library where a parallel I/O is strategy is adopted.

Acknowledgements

This work has been supported by the regional program of innovative actions PRAI-FESR Liguria.

References

1. J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon and A. White, *The Sourcebook of Parallel Computing*, Morgan Kaufmann, (2002).
2. R. Thakur, W. Gropp and E. Lusk, *Optimizing Noncontiguous Accesses in MPI-IO*, Parallel Computing, **28**, 83–105, (2002).
3. Parallel Virtual File System Version 2, see `http://www.pvfs.org`
4. Sun Microsystems Inc., *NFS: Network File System Version 3 Protocol Specification*, Sun Microsystems Inc., Mountain View, CA, (1993).
5. A. Ching, K. Coloma, J. Li and A. Choudhary, *High-performance techniques for parallel I/O*. In Handbook of Parallel Computing: Models, Algorithms, and Applications, CRC Press, (2007).
6. H. Yu and K.L. Ma, *A study of I/O methods for parallel visualization of large-scale data*, Parallel Computing, **31**, 167–183, (2005).
7. I.F. Sbalzarini, J.H.Walther, B. Polasek, P. Chatelain, M. Bergdorf, S.E. Hieber, E.M. Kotsalis, and P. Koumoutsakos, *A Software Framework for the Portable Parallelization of Particle-Mesh Simulations*, in Proceedings of Euro-Par 2006, LNCS **4128**, 730–739, (Springer, 2006).
8. Y. Zhu, H. Jiang, X. Qin and D. Swanson, *A Case Study of Parallel I/O for Biological sequence Search on Linux Clusters*, in: IEEE Proceeding of Cluster 2003, 308–315, (2003).
9. J. Li, W. Liao, A. Choudhary and V. Taylor, *I/O analysis and optimization for an AMR cosmology application*, in IEEE Proc. Cluster 2002, 119–126, (2002).
10. F. J. Seinstra and D. Koelma, *User transparency: a fully sequential programming model for efficient data parallel image processing*, Concurrency and Computation: Practice & Experience, **16**, 611–644, (2004).
11. J. M. Squyres, A. Lumsdaine and R. L. Stevenson, *A toolkit for parallel image processing*, in Parallel and Distributed Methods for Image Processing II, Proc. SPIE, vol **3452**, (1998).
12. P. Oliveira and H. du Buf, *SPMD image processing on Beowulf clusters: directives and libraries*, in: IEEE Proc. 7th IPDPS, (2003).
13. C. Nicolescu, P. Jonker, *EASY-PIPE an easy to use parallel image processing environment based on algorithmic skeletons*, in IEEE Proc. 15th IPDPS, (2001).
14. G. Ritter and J.Wilson, *Handbook of Computer Vision Algorithms in Image Algebra*, 2nd edition. (CRC Press, 2001).
15. A. Clematis, D. D'Agostino and A. Galizia, *An Object Interface for Interoperability of Image Processing Parallel Library in a Distributed Environment*, in Proc. ICIAP 2005, LNCS 3617, 584-591, (Springer, 2005).
16. ROMIO home page, `http://www.mcs.anl.gov/romio`